Midterm Exam

CMSC 433
Programming Language Technologies and Paradigms
Spring 2009

March 12, 2009

Guidelines

Put your name on each page before starting the exam. Write your answers directly on the exam sheets, using the back of the page as necessary. If you finish with more than 15 minutes left in the class, then bring your exam to the front when you are finished and leave the class as quietly as possible. Otherwise, please stay in your seat until the end.

If you have a question, raise your hand and I will come to you. Note, that I am unlikely to answer general questions however. If you feel an exam question assumes something that is not written, write it down on your exam sheet. Barring some unforeseen error on the exam, however, you shouldn’t need to do this at all, so be careful when making assumptions.

NOTE: There are quite a few questions to answer. Get started right away and budget your time wisely.

<table>
<thead>
<tr>
<th>Question</th>
<th>Points</th>
<th>Score</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>15</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>20</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>35</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>Total</td>
<td>100</td>
<td></td>
</tr>
</tbody>
</table>
1. Short answers (15 points). Give very short (1 to 2 sentences for each issue) answers to the following questions. **Longer responses to these questions will not be read.**

   (a) One key principle of design patterns is to "favor composition over inheritance." What does that mean? Give two reasons why this is beneficial.

   **Answer:**
   
   _To extend a class’ functionality, wrap it inside another class. Reuse original class’ functionality by delegating operations to the original class._
   
   _Removes implementation dependencies on base classes. Allows behaviors to change at runtime._

   (b) Information hiding is one criterion for modularizing software systems. What is information hiding? Provide one example from project 1 and describe specifically how information hiding is used there.

   **Answer:**
   
   _Information hiding means that modules encapsulate things that are likely to change._
   
   _P1 example: The Servlet interface hides the implementation of webserver functions._

   (c) In class we discussed the “uses” relation. Define what it means for a module A to use another module B. Consider a system that implements the Observer pattern. Class A implements the subject. Class B implements the Observers. When the subject changes, Class B instances are notified. Upon notification Class B instances update web documents to reflect the changed state of the subject. Does Class A “use” Class B.

   **Answer:**
   
   _Defn: A uses B when the correct functioning of A requires a correctly functioning instance of B._
   
   _No. Class A can operate correctly even if Class B does not._

   (d) In the context of software design, what is abstraction?

   **Answer:**
   
   _Decomposing a system into components should yield abstractions. Abstractions emphasize essential characteristics while suppressing implementation details._
2. The State Pattern (20 points). You are designing software for a vending machine. The vending machine has exactly one kind of product. The vending machine’s user interface supports two operations: insertCoin() and buy(). The insertCoin() method allows users to put money into the machine.

The insertCoin() method will either 1) accept the inserted coin if the amount of money currently inserted is less than that needed to buy an item. Otherwise it will ignore/return the inserted coin.

The buy() method indicates that the user is ready to buy one item of the product. Upon receiving a buy() request the vending machine, will either 1) dispense an item, give change and get ready for the next customer – if enough money has been inserted by the current customer, or 2) it will ignore the request – if enough money has not been inserted.

Use the state pattern to implement this system. Your solution will have two States: InsufficientFunds and SufficientFunds. The InsufficientFunds state indicates that there is currently not enough money to buy an item. The SufficientFunds state indicates that there currently is enough money to buy item.

```java
public interface State {
    State buy(VendingMachine vm);
    State insertCoin(VendingMachine vm, int c);
}

public class VendingMachine {
    State st; int itemCost = 20, depositedAmt=0;
    VendingMachine (State st) {this.st = st;}
    int getDepositedAmt() {return depositedAmt;}
    void setDepositedAmt (int amt) {depositedAmt = amt;}
    public int getItemCost () {return itemCost;}

    public void insertCoin(int c) {
        // FILL IN
    }

    public void buy() {
        // FILL IN
    }

    public static void main (String [] args) {
        VendingMachine vm = new VendingMachine(InsufficientFunds.instance);
        ....
    }
}
```
public class InsufficientFunds implements State {
    static InsufficientFunds instance = new InsufficientFunds();
    private InsufficientFunds() {};
    // FILL IN AS NEEDED

    public State insertCoin(VendingMachine vm, int amt) {
        System.out.println("Coin inserted");
        // FILL IN
    }

    public State buy(VendingMachine vm) {
        System.out.println("Product not dispensed.");
        // FILL IN
    }
}
public class SufficientFunds implements State {
    static SufficientFunds instance = new SufficientFunds();
    private SufficientFunds() {
        // FILL IN AS NEEDED
    }

    public State insertCoin(VendingMachine vm, int amt) {
        System.out.println("Coin ignored");
        // FILL IN
    }

    public State buy(VendingMachine vm) {
        System.out.println("Product dispensed.");
        // FILL IN
    }
}
public class VendingMachine {
    State st;
    int itemCost = 20, depositedAmt = 0;
    VendingMachine (State st) {this.st = st;}
    int getDepositedAmt() {return depositedAmt;}
    void setDepositedAmt (int amt) {depositedAmt = amt;}
    public int getItemCost () {return itemCost;}
    public void insertCoin(int c) {st = st.insertCoin(this, c);}
    public void buy() {st = st.buy(this);}
}

public class InsufficientFunds implements State {
    static InsufficientFunds instance = new InsufficientFunds();
    private InsufficientFunds() {};
    public State insertCoin(VendingMachine vm, int amt) {
        System.out.println("Coin inserted");
        vm.setDepositedAmt(vm.getDepositedAmt() + amt);
        if (vm.getDepositedAmt() >= vm.getItemCost()) {
            return SufficientFunds.instance;
        } else {
            return InsufficientFunds.instance;
        }
    }
    public State buy(VendingMachine vm) {
        System.out.println("Product not dispensed.");
        return InsufficientFunds.instance;
    }
}

public class SufficientFunds implements State {
    static SufficientFunds instance = new SufficientFunds();
    private SufficientFunds() {};
    public State insertCoin(VendingMachine vm, int amt) {
        System.out.println("Coin ignored");
        return SufficientFunds.instance;
    }
    public State buy(VendingMachine vm) {
        System.out.println("Product dispensed.");
        vm.setDepositedAmt(0);
        return InsufficientFunds.instance;
    }
}
3. Visitor Pattern (35 points). Implement 2 visitor classes for the following scenario. The FileSys
class models a File system. A File system has a single root Directory. Directories can contain
Directories and Files. Files must be contained within exactly one Directory. Except for the root
Directory, all Directories must be contained in exactly one Directory. The root Directory cannot
be contained in any Directory. Assume for the following problem that all File System objects are
correctly constructed.

Consider the driver program snippet below. It creates a File System rooted at “root”. The
directory “root” contains two directories, “c1” and “c2” and one file, “a”. Root/c1 contains one
file, “a”. The directory root/c2 is empty.

Dir root = new Dir("root");
Dir c1 = new Dir("c1");
root.add(c1);
Dir c2 = new Dir("c2");
root.add(c2);
File f1 = new File("a");
root.add(f1);
File f2 = new File("a");
c1.add(f2);

The File system is implemented with the following classes:

public interface FileSysVisitable {
    void accept (FileSysVisitor fsv);
}

class FileSys extends FileSysVisitable {
    String name;
    public abstract void accept(FileSysVisitor fsv);
    public String toString () {return name;}
}

class Dir extends FileSys {
    Dir parent;
    Set<FileSync> children = new HashSet<FileSync>();
    Dir(String name) {this.name = name;}
    public void add(FileSys fs) {children.add(fs);}
    public Set<FileSync> getChildren() {return children;}
    public void accept(FileSysVisitor fsv) {fsv.visit(this);}
}

class File extends FileSys {
    Dir parent;
    File (String name) {this.name = name;}
    public void accept(FileSysVisitor fsv) {fsv.visit(this);}
}

Your Visitor classes will implement the FileSysVisitor interface:

public interface FileSysVisitor{
    void visit(File f);
    void visit (Dir d);
}
(a) Your first Visitor is called QualifyVisitor. It must print out the fully qualified names (path from root directory to File System object) of every File and Directory in a given File System. For the example File System shown in the driver snippet, that output would be:

root/
root/c1/
root/c1/a
root/c2/
root/a

```java
public class QualifyVisitor implements FileSysVisitor {
    // FILL IN AS NEEDED

    public QualifyVisitor (String target) {
        // FILL IN
    }

    public void visit(File f) {
        // FILL IN
    }

    public void visit(Dir d) {
        // FILL IN
    }
}
```
public class QualifyVisitor implements FileSysVisitor {
    String QualName;

    public QualifyVisitor() {
        QualName = "";
    }

    public void visit(File f) {
        System.out.println(QualName + f);
    }

    public void visit(Dir d) {
        String tmpName = QualName;
        QualName += d + "/";
        System.out.println(QualName);
        for (FileSys child : d.getChildren()) {
            child.accept(this);
        }
        QualName = tmpName;
    }
}
(b) Your second Visitor is called SearchVisitor. It must determine whether a File System contains a File corresponding to a fully qualified File name (path from root directory to File object). For the example File System shown in the driver snippet, a search for “root/c1/a” should print out “true”, while a search for “root/c1/b” should print out “false”. Your solution should not do unnecessary work such as searching directories in which the file cannot be.

```java
public class SearchVisitor implements FileSysVisitor {
    String[] comps;
    // FILL IN AS NEEDED

    // comps contains path name components.
    // first path component stored at index 0.
    SearchVisitor(String target) {comps = target.split("/");}

    public void visit(File f) {
        // FILL IN
    }

    public void visit(Dir d) {
        // FILL IN
    }
}
```
public class SearchVisitor implements FileSysVisitor {
    String[] comps;
    int index = 0;
    boolean found = false;

    SearchVisitor(String target) {
        comps = target.split("/");
    }

    public void visit(File f) {
        if ((index == comps.length-1) && f.toString().equals(comps[index])){
            found = true;
        }
    }

    public void visit(Dir d) {
        if ((index < comps.length-1) && d.toString().equals(comps[index])) {
            index++;
            for (FileSys child : d.getChildren()) {
                child.accept(this);
                if (found) break;
            }
            index--;
        }
        if (index == 0) System.out.println(found);
    }
}
4. Software architecture (10 points).

Define architectural style? How are architectural styles different from architectural patterns and from domain-specific architectures (DSSAs)? Describe one architectural style (other than REST) and briefly discuss its main features/constraints.

Answer:

Architectural style: a named collection of architectural design decisions that are applicable in a given development context, constrain specific architectural design decisions and elicit beneficial qualities in the resulting system.

Architectural patterns tend to be more domain specific than architectural styles and tend to be focused on parts of a system rather than defining broad organizing principles for the entire system.

DSSAs are highly specialized to specific application domains where architectural styles are often useful across a variety of applications.

Example style: Interpreter. Interpreters are systems that execute commands/programs input to them. Interpreters allow a system to dynamically add functionality and can support code mobility.
5. Software architecture (10 points).

What are the 4 roles that architectural connectors fulfill? Define each. Give one specific example of a connector used in project 1. What role(s) did it play?

**Answer:**

- **Communication:** Transmit data.
- **Coordination:** Implement transfer of control.
- **Conversion:** Enable interaction between mismatched components.
- **Facilitation:** Optimize/streamline interactions.

*In project 1: Clients sent requests to the server via sockets. This played a communication role in the system. There were many procedure call connectors as well.*
6. Software architecture (10 points).

The REST architectural style is based on 3 key design choices: *Layered Separation*, *Replication* and *Limited common interfaces*. Briefly explain each of these choices and discuss what benefits/qualities they engender in a system.

**Answer:**

- **Layered Separation**: Components are layered to allow intermediary components to be added easily (proxies, filters, redirection services, etc.). This is done to improve scalability and support multiple system owners.

- **Replication**: Caching is highly encouraged. To support this REST encourages idempotent operations, representation metadata, and stateless interactions.

- **Limited common interfaces**: REST components support a very small number of general operations. This limits the complexity of client software and together with metadata allows new data types to be sent easily.